Longest Common Extensions in Sublinear Space

Philip Bille[1] ⋆, Inge Li Gørtz[1] ⋆, Mathias Bæk Tejs Knudsen[2] ⋆⋆,
Moshe Lewenstein[3] ⋆⋆⋆, and Hjalte Wedel Vildhøj[1]

[1] Technical University of Denmark, DTU Compute
[2] Department of Computer Science, University of Copenhagen
[3] Bar Ilan University

Abstract. The *longest common extension problem* (LCE problem) is to construct a data structure for an input string T of length n that supports $\mathrm{LCE}(i, j)$ queries. Such a query returns the length of the longest common prefix of the suffixes starting at positions i and j in T. This classic problem has a well-known solution that uses $\mathcal{O}(n)$ space and $\mathcal{O}(1)$ query time. In this paper we show that for any trade-off parameter $1 \leq \tau \leq n$, the problem can be solved in $\mathcal{O}(\frac{n}{\tau})$ space and $\mathcal{O}(\tau)$ query time. This significantly improves the previously best known time-space trade-offs, and almost matches the best known time-space product lower bound.

1 Introduction

Given a string T, the *longest common extension* of suffix i and j, denoted $\mathrm{LCE}(i, j)$, is the length of the longest common prefix of the suffixes of T starting at position i and j. The *longest common extension problem* (LCE problem) is to preprocess T into a compact data structure supporting fast longest common extension queries.

The LCE problem is a basic primitive that appears as a central subproblem in a wide range of string matching problems such as approximate string matching and its variations [1, 4, 13, 15, 21], computing exact or approximate repetitions [6, 14, 18], and computing palindromes [12, 19]. In many cases the LCE problem is the computational bottleneck.

Here we study the time-space trade-offs for the LCE problem, that is, the space used by the preprocessed data structure vs. the worst-case time used by LCE queries. The input string is given in read-only memory and is not counted in the space complexity. Throughout the paper we use ℓ as a shorthand for $\mathrm{LCE}(i, j)$. The standard trade-offs are as follows: At one extreme we can store

⋆ Supported by the Danish Research Council and the Danish Research Council under the Sapere Aude Program (DFF 4005-00267)

⋆⋆ Research partly supported by Mikkel Thorup's Advanced Grant from the Danish Council for Independent Research under the Sapere Aude research career programme and the FNU project AlgoDisc - Discrete Mathematics, Algorithms, and Data Structures.

⋆⋆⋆ This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development, and by a BSF grant 2010437

a suffix tree combined with an efficient nearest common ancestor (NCA) data structure [7,22]. This solution uses $\mathcal{O}(n)$ space and supports LCE queries in $\mathcal{O}(1)$ time. At the other extreme we do not store any data structure and instead answer queries simply by comparing characters from left-to-right in T. This solution uses $\mathcal{O}(1)$ space and answers an $\mathrm{LCE}(i,j)$ query in $\mathcal{O}(\ell) = \mathcal{O}(n)$ time. Recently, Bille et al. [2] presented a number of results. For a trade-off parameter τ, they gave: 1) a deterministic solution with $\mathcal{O}(\frac{n}{\tau})$ space and $\mathcal{O}(\tau^2)$ query time, 2) a randomized Monte Carlo solution with $\mathcal{O}(\frac{n}{\tau})$ space and $\mathcal{O}(\tau \log(\frac{\ell}{\tau})) = \mathcal{O}(\tau \log(\frac{n}{\tau}))$ query time, where all queries are correct with high probability, and 3) a randomized Las Vegas solution with the same bounds as 2) but where all queries are guaranteed to be correct. Bille et al. [2] also gave a lower bound showing that any data structure for the LCE problem must have a time-space product of $\Omega(n)$ bits.

Our Results Let τ be a trade-off parameter. We present four new solutions with the following improved bounds. Unless otherwise noted the space bound is the number of words on a standard RAM with logarithmic word size, not including the input string, which is given in read-only memory.

– A deterministic solution with $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau \log^2(n/\tau))$ query time.
– A randomized Monte Carlo solution with $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ query time, such that all queries are correct with high probability.
– A randomized Las Vegas solution with $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ query time.
– A derandomized version of the Monte Carlo solution with $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ query time.

Hence, we obtain the first trade-off for the LCE problem with a linear time-space product in the full range from constant to linear space. This almost matches the

	Data Structure			**Preprocessing**		**Trade-off range**	**Reference**
	Space	Query	Correct	Space	Time		
Deterministic	1	ℓ	always	1	1	-	Store nothing
	n	1	always	n	n	-	Suffix tree + NCA
	$\frac{n}{\tau}$	τ^2	always	$\frac{n}{\tau}$	$\frac{n^2}{\tau}$	$1 \leq \tau \leq \sqrt{n}$	[2]
	$\frac{n}{\tau}$	$\tau \log^2 \frac{n}{\tau}$	always	$\frac{n}{\tau}$	n^2	$\frac{1}{\log n} \leq \tau \leq n$	this paper, Sec. 2
	$\frac{n}{\tau}$	τ	always	$\frac{n}{\tau}$	$n^{2+\varepsilon}$	$1 \leq \tau \leq n$	this paper, Sec. 4
Monte Carlo	$\frac{n}{\tau}$	$\tau \log \frac{\ell}{\tau}$	w.h.p.	$\frac{n}{\tau}$	n	$1 \leq \tau \leq n$	[2]
	$\frac{n}{\tau}$	τ	w.h.p.	$\frac{n}{\tau}$	$n \log \frac{n}{\tau}$	$1 \leq \tau \leq n$	this paper, Sec. 3
Las Vegas	$\frac{n}{\tau}$	$\tau \log \frac{\ell}{\tau}$	always	$\frac{n}{\tau}$	$n(\tau + \log n)$ w.h.p.	$1 \leq \tau \leq n$	[2]
	$\frac{n}{\tau}$	τ	always	$\frac{n}{\tau}$	$n^{3/2}$ w.h.p.	$1 \leq \tau \leq n$	this paper, Sec. 3.5

Table 1. Overview of solutions for the LCE problem. Here $\ell = \mathrm{LCE}(i,j)$, $\varepsilon > 0$ is an arbitrarily small constant and w.h.p. (with high probability) means with probability at least $1 - n^{-c}$ for an arbitrarily large constant c. The data structure is *correct* if it answers all LCE queries correctly.

time-space product lower bound of $\Omega(n)$ *bits*, and improves the best deterministic upper bound by a factor of τ, and the best randomized bound by a factor $\log(\frac{n}{\tau})$. See the columns marked *Data Structure* in Table 1 for a complete overview.

While our main focus is the space and query time complexity, we also provide efficient *preprocessing* algorithms for building the data structures, supporting independent trade-offs between the preprocessing time and preprocessing space. See the columns marked *Preprocessing* in Table 1.

To achieve our results we develop several new techniques and specialized data structures which are likely of independent interest. For instance, in our deterministic solution we develop a novel recursive decomposition of LCE queries and for the randomized solution we develop a new sampling technique for Karp-Rabin fingerprints that allow fast LCE queries. We also give a general technique for efficiently derandomizing algorithms that rely on "few" or "short" Karp-Rabin fingerprints, and apply the technique to derandomize our Monte Carlo algorithm. To the best of our knowledge, this is the first derandomization technique for Karp-Rabin fingerprints.

Preliminaries We assume an integer alphabet, i.e., T is chosen from some alphabet $\Sigma = \{0, \dots, n^c\}$ for some constant c, so every character of T fits in $\mathcal{O}(1)$ words. For integers $a \leq b$, $[a, b]$ denotes the range $\{a, a+1, \dots, b\}$ and we define $[n] = [0, n-1]$. For a string $S = S[1]S[2]\dots S[|S|]$ and positions $1 \leq i \leq j \leq |S|$, $S[i...j] = S[i]S[i+1]\cdots S[j]$ is a *substring* of length $j - i + 1$, $S[i...] = S[i, |S|]$ is the i^{th} *suffix* of S, and $S[...i] = S[1, i]$ is the i^{th} *prefix* of S.

2 Deterministic Trade-Off

Here we describe a completely deterministic trade-off for the LCE problem with $\mathcal{O}(\frac{n}{\tau}\log\frac{n}{\tau})$ space and $\mathcal{O}(\tau\log\frac{n}{\tau})$ query time for any $\tau \in [1, n]$. Substituting $\hat{\tau} = \tau/\log(n/\tau)$, we obtain the bounds reflected in Table 1 for $\hat{\tau} \in [1/\log n, n]$.

A key component in this solution is the following observation that allows us to reduce an $\mathrm{LCE}(i, j)$ query on T to another query $\mathrm{LCE}(i', j')$ where i' and j' are both indices in either the first or second half of T.

Observation 1 *Let i, j and j' be indices of T, and suppose that* $\mathrm{LCE}(j', j) \geq \mathrm{LCE}(i, j)$. *Then* $\mathrm{LCE}(i, j) = \min(\mathrm{LCE}(i, j'), \mathrm{LCE}(j', j))$.

We apply Observation 1 recursively to bring the indices of the initial query within distance τ in $\mathcal{O}(\log(n/\tau))$ rounds. We show how to implement each round with a data structure using $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ time. This leads to a solution using $\mathcal{O}(\frac{n}{\tau}\log\frac{n}{\tau})$ space and $\mathcal{O}(\tau\log\frac{n}{\tau})$ query time. Finally in Section 2.4, we show how to efficiently solve the LCE problem for indicies within distance τ in $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ time by exploiting periodicity properties of LCEs.

2.1 The Data Structure

We will store several data structures, each responsible for a specific subinterval $I = [a, b] \subseteq [1, n]$ of positions of the input string T. Let $I_{\text{left}} = [a, (a+b)/2]$,

$I_{\text{right}} = \ ((a+b)/2, b]$, and $|I| = b - a + 1$. The task of the data structure for I will be to reduce an $\text{LCE}(i, j)$ query where $i, j \in I$ to one where both indices belong to either I_{left} or I_{right}.

The data structure stores information for $\mathcal{O}(|I|/\tau)$ suffixes of T that start in I_{right}. More specifically, we store information for the suffixes starting at positions $b - k\tau \in I_{\text{right}}$, $k = 0, 1, \ldots, (|I|/2)/\tau$. We call these the *sampled positions* of I_{right}. See Figure 1 for an illustration.

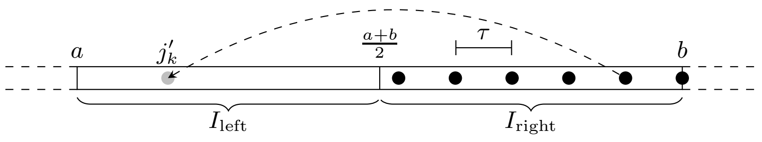

Fig. 1. Illustration of the contents of the data structure for the interval $I = [a, b]$. The black dots are the sampled positions in I_{right}, and each such position has a pointer to an index $j'_k \in I_{\text{left}}$.

For every sampled position $b - k\tau \in I_{\text{right}}$, $k = 0, 1, \ldots, (|I|/2)/\tau$, we store the index j'_k of the suffix starting in I_{left} that achieves the maximum LCE value with the suffix starting at the sampled position, i.e., $T[b - k\tau...]$ (ties broken arbitrarily). Along with j'_k, we also store the value of the LCE between suffix $T[j'_k...]$ and $T[b - k\tau...]$. Formally, j'_k and L_k are defined as follows,

$$j'_k = \underset{h \in I_{\text{left}}}{\operatorname{argmax}} \text{LCE}(h, b - k\tau) \quad \text{and} \quad L_k = \text{LCE}(j'_k, b - k\tau) \ .$$

Building the structure We construct the above data structure for the interval $[1, n]$, and build it recursively for $[1, n/2]$ and $(n/2, n]$, stopping when the length of the interval becomes smaller than τ.

2.2 Answering a Query

We now describe how to reduce a query $\text{LCE}(i, j)$ where $i, j \in I$ to one where both indices are in either I_{left} or I_{right}. Suppose without loss of generality that $i \in I_{\text{left}}$ and $j \in I_{\text{right}}$. We start by comparing $\delta < \tau$ pairs of characters of T, starting with $T[i] = T[j]$, until 1) we encounter a mismatch, 2) both positions are in I_{right} or 3) we reach a sampled position in I_{right}. It suffices to describe the last case, in which $T[i, i+\delta] = T[j, j+\delta]$, $i + \delta \in I_{\text{left}}$ and $j + \delta = b - k\tau \in I_{\text{right}}$ for some k. Then by Observation 1, we have that

$$\begin{aligned}\text{LCE}(i, j) &= \delta + \text{LCE}(i+\delta, j+\delta) \\ &= \delta + \min(\text{LCE}(i+\delta, j'_k), \text{LCE}(j'_k, b - k\tau)) \\ &= \delta + \min(\text{LCE}(i+\delta, j'_k), L_k) \ .\end{aligned}$$

Thus, we have reduced the original query to computing the query $\mathrm{LCE}(i+\delta, j'_k)$ in which both indices are in I_{left}.

2.3 Analysis

Each round takes $\mathcal{O}(\tau)$ time and halves the upper bound for $|i-j|$, which initially is n. Thus, after $\mathcal{O}(\tau \log(n/\tau))$ time, the initial LCE query has been reduced to one where $|i-j| \leq \tau$. At each of the $\mathcal{O}(\log(n/\tau))$ levels, the number of sampled positions is $(n/2)/\tau$, so the total space used is $\mathcal{O}((n/\tau)\log(n/\tau))$.

2.4 Queries with Nearby Indices

We now describe the data structure used to answer a query $\mathrm{LCE}(i,j)$ when $|i-j| \leq \tau$. We first give some necessary definitions and properties of periodic strings. We say that the integer $1 \leq p \leq |S|$ is a *period* of a string S if any two characters that are p positions apart in S match, i.e., $S[i] = S[i+p]$ for all positions i s.t. $1 \leq i < i+p \leq |S|$. The following is a well-known property of periods.

Lemma 1 (Fine and Wilf [5]). *If a string S has periods a and b and $|S| \geq |a| + |b| - \gcd(a,b)$, then $\gcd(a,b)$ is also a period of S.*

The period of S is the smallest period of S and we denote it by $\mathrm{per}(S)$. If $\mathrm{per}(S) \leq |S|/2$, we say S is *periodic*. A periodic string S might have many periods smaller than $|S|/2$, however it follows from the above lemma that

Corollary 1. *All periods smaller than $|S|/2$ are multiples of $\mathrm{per}(S)$.*

The Data Structure Let $T_k = T[k\tau...(k+2)\tau - 1]$ denote the substring of length 2τ starting at position $k\tau$ in T, $k = 0, 1, \ldots, n/\tau$. For the strings T_k that are periodic, let $p_k = \mathrm{per}(T_k)$ be the period. For every periodic T_k, the data structure stores the length ℓ_k of the maximum substring starting at position $k\tau$, which has period p_k. Nothing is stored if T_k is aperiodic.

Answering a Query We may assume without loss of generality that $i = k\tau$, for some integer k. If not, then we check whether $T[i+\delta] = T[j+\delta]$ until $i+\delta = k\tau$. Hence, assume that $i = k\tau$ and $j = i + d$ for some $0 < d \leq \tau$. In $\mathcal{O}(\tau)$ time, we first check whether $T[i+\delta] = T[j+\delta]$ for all $\delta \in [0, 2\tau]$. If we find a mismatch we are done, and otherwise we return $\mathrm{LCE}(i,j) = \ell_k - d$.

Correctness If a mismatch is found when checking that $T[i+\delta] = T[j+\delta]$ for all $\delta \in [0, 2\tau]$, the answer is clearly correct. Otherwise, we have established that $d \leq \tau$ is a period of T_k, so T_k is periodic and d is a multiple of p_k (by Corollary 1). Consequently, $T[i+\delta] = T[j+\delta]$ for all δ s.t. $d + \delta \leq \ell_k$, and thus $\mathrm{LCE}(i,j) = \ell_k - d$.

2.5 Preprocessing

The preprocessing of the data structure is split into the preprocessing of the recursive data structure for queries where $|i-j| > \tau$ and the preprocessing of the data structure for nearby indices, i.e., $|i-j| \leq \tau$.

If we allow $\tilde{\mathcal{O}}(n)$ space during the preprocessing phase then both can be solved in $\tilde{\mathcal{O}}(n)$ time. If we insist on $\tilde{\mathcal{O}}(n/\tau)$ space during preprocessing then the times are not as good.

Recursive Data Structure Say $\tilde{\mathcal{O}}(n)$ space is allowed during preprocessing. Generate a 2D range searching environment with a point $(x(l), y(l))$ for every text index $l \in [1, n]$. Set $x(l) = l$ and $y(l)$ equal to the rank of suffix $T[l...]$ in the lexicographical sort of the suffixes. To compute $j'_k \in I_{\text{left}} = [a, (a+b)/2]$ that has the maximal LCE value with a sampled position $i = b - k\tau \in I_{\text{right}} = ((a+b)/2, b]$ we require two 3-sided range successor queries, see detailed definitions in [17]. The first is a range $[a, (a+b)/2] \times (-\infty, y(i) - 1]$ which returns the point within the range with y-coordinate closest to, and less-than, $y(i)$. The latter is a range $[a, (a+b)/2] \times [y(i) + 1, \infty)$ with point with y-coordinate closest to, and greater-than, $y(i)$ returned.

Let l' and l'' be the x-coordinates, i.e., text indices, of the points returned by the queries. By definition both are in $[a, (a+b)/2]$. Among suffixes starting in $[a, (a+b)/2]$, $T[l'...]$ has the largest rank in the lexicographic sort which is less than l. Likewise, $T[l''...]$ has the smallest rank in the lexicographic sort which is greater than l. Hence, j'_k is equal to either l' or l'', and the larger of LCE(l, l') and LCE(l, l'') determines which one.

The LCE computation can be implemented with standard suffix data structures in $\mathcal{O}(n)$ space and $\mathcal{O}(1)$ query time for an overall $\mathcal{O}(n)$ time. 3-sided range successor queries can be preprocessed in $\mathcal{O}(n \log n)$ space and time and queries can be answered in $\mathcal{O}(\log \log n)$ time, see [11, 16] for an overall $\mathcal{O}(n \log \log n)$ time and $\mathcal{O}(n \log n)$ space.

If one desires to keep the space limited to $\tilde{\mathcal{O}}(n/\tau)$ space then generate a sparse suffix array for the (evenly spaced) suffixes starting at $k\tau$ in I_{right}, $k = 0, 1, \ldots, (|I|/2)/\tau$, see [9]. Now for every suffix in I_{left} find its location in the sparse suffix array. From this data one can compute the desired answer. The time to compute the sparse suffix array is $\mathcal{O}(n)$ and the space is $\mathcal{O}(n/\tau)$. The time to search a suffix in I_{left} is $\mathcal{O}(n + \log(|I|/\tau))$ using the classical suffix search in the suffix array [20]. The time per I is $\mathcal{O}(|I|n)$. Overall levels this yields $\mathcal{O}(n \cdot n + n \cdot (n/2) + n \cdot (n/4) + \cdots + n \cdot 1) = \mathcal{O}(n^2)$ time.

The Nearby Indices Data Structure If T_k is periodic then we need to find its period p_k and find the length ℓ_k of the maximum substring starting at position $k\tau$, which has period p_k.

Finding period p_k, if it exists, in T_k can be done in $\mathcal{O}(\tau)$ time and constant space using the algorithm of [3]. The overall time is $\mathcal{O}(n)$ and the overall space is $\mathcal{O}(n/\tau)$ for the p_k's.

To compute ℓ_k we check whether the period p_k of T_k extends to T_{k+2}. That is whether p_k is the period of $T[k\tau ... (k+4)\tau - 1]$. The following is why we do so.

Lemma 2. *Suppose T_k and T_{k+2} are periodic with periods p_k and p_{k+2}, respectively. If $T_k \cdot T_{k+2}$ is periodic with a period of length at most τ then p_k is the period and p_{k+2} is a rotation of p_k.*

Proof. To reach a contradition, suppose that p_k is not the period of $T_k \cdot T_{k+2}$, i.e., it has period p' of length at most τ. Since T_k is of size 2τ, p' must also be a period of T_k. However, p_k and p' are both prefixes of T_k and, hence, must have different lengths. By Corollary 1 $|p'|$ must be a multiple of $|p_k|$ and, hence, not a period. The fact that p_{k+2} is a rotation of p_k can be deduced similarly. $\square$

By induction it follows that $T_k \cdot T_{k+2} \cdot T_{k+4} \cdot \ldots \cdot T_{k+2M}$, $M \geq 1$, is periodic with p_k if each concatenated pair of $T_{k+2i} \cdot T_{k+2(i+1)}$ is periodic with a period at most τ. This suggests the following scheme. Generate a binary array of length n/τ where location j is 1 iff $T_{k+2(j-1)} \cdot T_{k+2j}$ has a period of length of at most τ. In all places where there is a 0, i.e., $T_k \cdot T_{k+2}$ does not have a period of length at most τ and T_k does have period p_k, we have that $\ell_k \in [2\tau, 4\tau - 1]$. We directly compute ℓ_k by extending p_k in $T_k \cdot T_{k+2}$ as far as possible. Now, using the described array and the $\ell_k \in [2\tau, 4\tau - 1]$, in a single sweep for the even $k\tau$'s (and a seperate one for the odd ones) from right to left we can compute all ℓ_k's.

It is easy to verify that the overall time is $\mathcal{O}(n)$ and the space is $\mathcal{O}(n/\tau)$.

3 Randomized Trade-Offs

In this section we describe a randomized LCE data structure using $\mathcal{O}(n/\tau)$ space with $\mathcal{O}(\tau + \log \frac{\ell}{\tau})$ query time. In Section 3.6 we describe another $\mathcal{O}(n/\tau)$-space LCE data structure that either answers an LCE query in constant time, or provides a certificate that $\ell \leq \tau^2$. Combining the two data structures, shows that the LCE problem can be solved in $\mathcal{O}(n/\tau)$ space and $\mathcal{O}(\tau)$ time.

The randomization comes from our use of Karp-Rabin fingerprints [10] for comparing substrings of T for equality. Before describing the data structure, we start by briefly recapping the most important definitions and properties of Karp-Rabin fingerprints.

3.1 Karp-Rabin Fingerprints

For a prime p and $x \in [p]$ the Karp-Rabin fingerprint [10], denoted $\phi_{p,x}(T[i...j])$, of the substring $T[i...j]$ is defined as

$$\phi_{p,x}(T[i...j]) = \sum_{i \leq k \leq j} T[k] x^{k-i} \bmod p \, .$$

If $T[i...j] = T[i'...j']$ then clearly $\phi_{p,x}(T[i...j]) = \phi_{p,x}(T[i'...j'])$. In the Monte Carlo and the Las Vegas algorithms we present we will choose p such that $p =$

$\Theta(n^{4+c})$ for some constant $c > 0$ and x uniformly from $[p]\setminus\{0\}$. In this case a simple union bound shows that the converse is also true with high probability, i.e., ϕ is *collision-free* on all substring pairs of T with probability at least $1-n^{-c}$. Storing a fingerprint requires $\mathcal{O}(1)$ space. When p, x are clear from the context we write $\phi = \phi_{p,x}$.

For shorthand we write $f(i) = \phi(T[1,i]), i \in [1,n]$ for the fingerprint of the i^{th} prefix of T. Assuming that we store the exponent $x^i \mod p$ along with the fingerprint $f(i)$, the following two properties of fingerprints are well-known and easy to show.

Lemma 3. *1) Given $f(i)$, the fingerprint $f(i \pm a)$ for some integer a, can be computed in $\mathcal{O}(a)$ time. 2) Given fingerprints $f(i)$ and $f(j)$, the fingerprint $\phi(T[i..j])$ can be computed in $\mathcal{O}(1)$ time.*

In particular this implies that for a fixed length l, the fingerprint of all substrings of length l of T can be enumerated in $\mathcal{O}(n)$ time using a sliding window.

3.2 Overview

The main idea in our solution is to binary search for the $\text{LCE}(i,j)$ value using Karp-Rabin fingerprints. Suppose for instance that $\phi(T[i, i+M]) \neq \phi(T[j, j+M])$ for some integer M, then we know that $\text{LCE}(i,j) \leq M$, and thus we can find the true $\text{LCE}(i,j)$ value by comparing $\log(M)$ additional pair of fingerprints. The challenge is to obtain the fingerprints quickly when we are only allowed to use $\mathcal{O}(n/\tau)$ space. We will partition the input string T into n/τ *blocks* each of length τ. Within each block we sample a number of equally spaced positions. The data structure consists of the fingerprints of the prefixes of T that ends at the sampled positions, i.e., we store $f(i)$ for all sampled positions i. In total we sample $\mathcal{O}(n/\tau)$ positions. If we just sampled a single position in each block (similar to the approach in [2]), we could compute the fingerprint of any substring in $\mathcal{O}(\tau)$ time (see Lemma 3), and the above binary search algorithm would take time $\mathcal{O}(\tau \log n)$ time. We present a new sampling technique that only samples an additional $\mathcal{O}(n/\tau)$ positions, while improving the query time to $\mathcal{O}(\tau + \log(\ell/\tau))$.

Preliminary definitions We partition the input string T into n/τ blocks of τ positions, and by *block k* we refer to the positions $[k\tau, k\tau+\tau)$, for $k \in [n/\tau]$.

We assume without loss of generality that n and τ are both powers of two. Every position $q \in [1,n]$ can be represented as a bit string of length $\lg n$. Let $q \in [1,n]$ and consider the binary representation of q. We define the leftmost $\lg(n/\tau)$ bits and rightmost $\lg(\tau)$ bits to be the *head*, denoted $h(q)$ and the *tail*, denoted $t(q)$, respectively. A position is *block aligned* if $t(q) = 0$. The *significance* of q, denoted $s(q)$, is the number of trailing zeros in $h(q)$. Note that the τ positions in any fixed block $k \in [n/\tau]$ all have the same head, and thus also the same significance, which we denote by μ_k. See Figure 2.

$$q \quad \overbrace{0\,1\,1\,0\,0\,1\,\underbrace{0\,0\,0\,0\,0}_{s(q)}}^{h(q)}\overbrace{1\,1\,1\,0\,1\,0\,1\,1}^{t(q)}$$

Fig. 2. Example of the definitions for the position $q = 205035$ in a string of length $n = 2^{19}$ with block length $\tau = 2^8$. Here $h(q)$ is the first $\lg(n/\tau) = 11$ bits, and $t(q)$ is the last $\lg(\tau) = 8$ bits in the binary representation of q. The significance is $s(q) = 5$.

3.3 The Monte Carlo Data Structure

The data structure consists of the values $f(i)$, $i \in \mathcal{S}$, for a specific set of sampled positions $\mathcal{S} \subseteq [1, n]$, along with the information necessary in order to look up the values in constant time. We now explain how to construct the set $\mathcal{S}$. In block $k \in [n/\tau]$ we will sample $b_k = \min\left\{2^{\lfloor \mu_k/2 \rfloor}, \tau\right\}$ evenly spaced positions, where μ_k is the significance of the positions in block k, i.e., $\mu_k = s(k\tau)$. More precisely, in block k we sample the positions $\mathcal{B}_k = \{k\tau + j\tau/b_k \mid j \in [b_k]\}$, and let $\mathcal{S} = \cup_{k \in [n/\tau]} \mathcal{B}_k$. See Figure 3.

We now bound the size of $\mathcal{S}$. The significance of a block is at most $\lg(n/\tau)$, and there are exactly $2^{\lg(n/\tau)-\mu}$ blocks with significance μ, so

$$|\mathcal{S}| = \sum_{k=0}^{n/\tau-1} b_k \le \sum_{\mu=0}^{\lg(n/\tau)} 2^{\lg(n/\tau)-\mu} 2^{\lfloor \mu/2 \rfloor} \le \frac{n}{\tau} \sum_{\mu=0}^{\infty} 2^{-\mu/2} = \left(2 + \sqrt{2}\right) \frac{n}{\tau} = \mathcal{O}\left(\frac{n}{\tau}\right).$$

k	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
μ_k	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0
b_k	4.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0

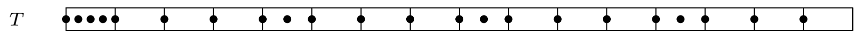

Fig. 3. Illustration of a string T partitioned into 16 blocks each of length τ. The significance μ_k for the positions in each block $k \in [n/\tau]$ is shown, as well as the b_k values. The block dots are the sampled positions $\mathcal{S}$.

3.4 Answering a query

We now describe how to answer an LCE(i, j) query. We will assume that i is block aligned, i.e., $i = k\tau$ for some $k \in [n/\tau]$. Note that we can always obtain this situation in $\mathcal{O}(\tau)$ time by initially comparing at most $\tau - 1$ pairs of characters of the input string directly.

Algorithm 1 shows the query algorithm. It performs an exponential search to locate the block in which the first mismatch occurs, after which it scans the block directly to locate the mismatch. The search is performed by calls to `check`(i, j, c), which computes and compares $\phi(T[i...i+c])$ and $\phi(T[j...j+c])$. In other words, assuming that ϕ is collision-free, `check`(i,j,c) returns `true` if $\mathrm{LCE}(i,j) \geq c$ and `false` otherwise.

Algorithm 1 Computing the answer to a query $\mathrm{LCE}(i,j)$

```
1:  procedure LCE($i, j$)
2:      $\hat{\ell} \leftarrow 0$
3:      $\mu \leftarrow 0$
4:      while check($i, j, 2^\mu\tau$) do        ▷ Compute an interval such that $\ell \in [\hat{\ell}, 2\hat{\ell}]$.
5:          $(i, j, \hat{\ell}) \leftarrow (i + 2^\mu\tau, j + 2^\mu\tau, \hat{\ell} + 2^\mu\tau)$
6:          if $s(j) > \mu$ then
7:              $\mu \leftarrow \mu + 1$
8:      while $\mu > 0$ do        ▷ Identify the block in which the first mismatch occurs
9:          if check($i, j, 2^{\mu-1}\tau$) then
10:             $(i, j, \hat{\ell}) \leftarrow (i + 2^{\mu-1}\tau, j + 2^{\mu-1}\tau, \hat{\ell} + 2^{\mu-1}\tau)$
11:         $\mu \leftarrow \mu - 1$
12:     while $T[i] = T[j]$ do  ▷ Scan the final block left to right to find the mismatch
13:         $(i, j, \hat{\ell}) \leftarrow (i + 1, j + 1, \hat{\ell} + 1)$
14:     return $\hat{\ell}$
```

Analysis We now prove that Algorithm 1 correctly computes $\ell = \mathrm{LCE}(i,j)$ in $\mathcal{O}(\tau + \log(\ell/\tau))$ time. The algorithm is correct assuming that `check`$(i,j,2^\mu\tau)$ always returns the correct answer, which will be the case if ϕ is collision-free.

The following is the key lemma we need to bound the time complexity.

Lemma 4. *Throughout Algorithm 1 it holds that $\ell \geq (2^\mu - 1)\,\tau$, $s(j) \geq \mu$, and μ is increased in at least every second iteration of the first* **while**-*loop.*

Proof. We first prove that $s(j) \geq \mu$. The claim holds initially. In the first loop j is changed to $j + 2^\mu\tau$, and $s(j + 2^\mu\tau) \geq \min\{s(j), s(2^\mu\tau)\} = \min\{s(j), \mu\} = \mu$, where the last equality follows from the induction hypothesis $s(j) \geq \mu$. Moreover, μ is only incremented when $s(j) > \mu$. In the second loop j is changed to $j+2^{\mu-1}\tau$, which under the assumption that $s(j) \geq \mu$, has significance $s(j+2^{\mu-1}\tau) = \mu-1$. Hence the invariant is restored when μ is decremented at line 11.

Now consider an iteration of the first loop where μ is not incremented, i.e., $s(j) = \mu$. Then $\frac{j}{2^\mu\tau}$ is an odd integer, i.e. $\frac{j+2^\mu\tau}{2^\mu\tau}$ is even, and hence $s(j+2^\mu\tau) > \mu$, so μ will be incremented in the next iteration of the loop.

In order to prove that $\ell \geq (2^\mu - 1)\,\tau$ we will prove that $\hat{\ell} \geq (2^\mu - 1)\,\tau$ in the first loop. This is trivial by induction using the observation that $(2^\mu - 1)\,\tau + 2^\mu\tau = \left(2^{\mu+1} - 1\right)\tau$. □

Since $\ell \geq (2^\mu - 1)\tau$ and μ is increased at least in every second iteration of the first loop and decreased in every iteration of the second loop, it follows that there are $\mathcal{O}(\log(\ell/\tau))$ iterations of the two first loops. The last loop takes $\mathcal{O}(\tau)$ time. It remains to prove that the time to evaluate the $\mathcal{O}(\log(\ell/\tau))$ calls to $\texttt{check}(i, j, 2^\mu\tau)$ sums to $\mathcal{O}(\tau + \log(\ell/\tau))$.

Evaluating $\texttt{check}(i, j, 2^\mu\tau)$ requires computing $\phi(T[i...i+2^\mu\tau])$ and $\phi(T[j...j+2^\mu\tau])$. The first fingerprint can be computed in constant time because i and $i+2^\mu\tau$ are always block aligned (see Lemma 3). The time to compute the second fingerprint depends on how far j and $j+2^\mu\tau$ each are from a sampled position, which in turn depends inversely on the significance of the block containing those positions. By Lemma 4, μ is always a lower bound on the significance of j, which implies that μ also lower bounds the significance of $j+2^\mu\tau$, and thus by the way we sample positions, neither will have distance more than $\tau/2^{\lfloor\mu/2\rfloor}$ to a sampled position in $\mathcal{S}$. Finally, note that by the way μ is increased and decreased, $\texttt{check}(i, j, 2^\mu\tau)$ is called at most three times for any fixed value of μ. Hence, the total time to compute all necessary fingerprints can be bounded as

$$\mathcal{O}\left(\sum_{\mu=0}^{\lg(\ell/\tau)} 1 + \tau/2^{\lfloor\mu/2\rfloor}\right) = \mathcal{O}(\tau + \log(\ell/\tau)) \ .$$

3.5 The Las Vegas Data Structure

We now describe an $\mathcal{O}(n^{3/2})$-time and $\mathcal{O}(n/\tau)$-space algorithm for verifiying that ϕ is *collision-free* on all pairs of substrings of T that the query algorithm compares. If a collision is found we pick a new ϕ and try again. With high probability we can find a collision-free ϕ in a constant number of trials, so we obtain the claimed Las Vegas data structure.

If $\tau \le \sqrt{n}$ we use the verification algorithm of Bille et al. [2], using $\mathcal{O}(n\tau + n\log n)$ time and $\mathcal{O}(n/\tau)$ space. Otherwise, we use the simple $\mathcal{O}(n^2/\tau)$-time and $\mathcal{O}(n/\tau)$-space algorithm described below.

Recall that all fingerprint comparisions in our algorithm are of the form

$$\phi\big(T[k\tau...k\tau + 2^l\tau - 1]\big) \overset{?}{=} \phi\big(T[j...j + 2^l\tau - 1]\big)$$

for some $k \in [n/\tau], j \in [n], l \in [\log(n/\tau)]$. The algorithm checks each $l \in [\log(n/\tau)]$ separately. For a fixed l it stores the fingerprints $\phi(T[k\tau...k\tau + 2^l\tau])$ for all $k \in [n/\tau]$ in a hash table $\mathcal{H}$. This can be done in $\mathcal{O}(n)$ time and $\mathcal{O}(n/\tau)$ space. For every $j \in [n]$ the algorithm then checks whether $\phi\big(T[j...j+2^l\tau]\big) \in \mathcal{H}$, and if so, it verifies that the underlying two substrings are in fact the same by comparing them character by character in $\mathcal{O}(2^l\tau)$ time. By maintaining the fingerprint inside a sliding window of length $2^l\tau$, the verification time for a fixed l becomes $\mathcal{O}(n2^l\tau)$, i.e., $\mathcal{O}(n^2/\tau)$ time for all $l \in [\log(n/\tau)]$.

3.6 Queries with Long LCEs

In this section we describe an $\mathcal{O}(\frac{n}{\tau})$ space data structure that in constant time either correctly computes $\text{LCE}(i, j)$ or determines that $\text{LCE}(i, j) \le \tau^2$. The data structure can be constructed in $\mathcal{O}(n \log \frac{n}{\tau})$ time by a Monte Carlo or Las Vegas algorithm.

The Data Structure Let $\mathcal{S}_\tau \subseteq [1,n]$ called the *sampled positions* of T (to be defined below), and consider the sets A and B of suffixes of T and T^R, respectively.

$$A = \{T[i...] \mid i \in \mathcal{S}_\tau\} \quad , \quad B = \{T[...i]^R \mid i \in \mathcal{S}_\tau\} \ .$$

We store a data structure for A and B, that allows us to perform constant time longest common extension queries on any pair of suffixes in A or any pair in B. This can be achieved by well-known techniques, e.g., storing a sparse suffix tree for A and B, equipped with a nearest common ancestor data structure. To define $\mathcal{S}_\tau$, let $D_\tau = \{0, 1, \ldots, \tau\} \cup \{2\tau, \ldots, (\tau-1)\tau\}$, then

$$\mathcal{S}_\tau = \{1 \leq i \leq n \ \mid \ i \mod \tau^2 \in D_\tau\} \ . \tag{1}$$

Answering a Query To answer an LCE query, we need the following definitions. For $i, j \in \mathcal{S}_\tau$ let $\mathrm{LCE}_R(i,j)$ denote the longest common prefix of $T[...i]^R \in B$ and $T[...j]^R \in B$. Moreover, for $i, j \in [n]$, we define the function

$$\delta(i,j) = \big(((i-j) \mod \tau) - i\big) \mod \tau^2 \ . \tag{2}$$

We will write δ instead of $\delta(i,j)$ when i and j are clear from the context.

The following lemma gives the key property that allows us to answer a query.

Lemma 5. *For any $i, j \in [n - \tau^2]$, it holds that $i + \delta, j + \delta \in \mathcal{S}_\tau$.*

Proof. Direct calculation shows that $(i+\delta) \mod \tau^2 \leq \tau$, and that $(j+\delta) \mod \tau = 0$, and thus by definition both $i+\delta$ and $j+\delta$ are in $\mathcal{S}_\tau$.

To answer a query $\mathrm{LCE}(i,j)$, we first verify that $i, j \in [n-\tau^2]$ and that $\mathrm{LCE}_R(i+\delta, j+\delta) \geq \delta$. If this is not the case, we have established that $\mathrm{LCE}(i,j) \leq \delta < \tau^2$, and we stop. Otherwise, we return $\delta + \mathrm{LCE}(i+\delta, j+\delta) - 1$.

Analysis To prove the correctness, suppose $i, j \in [n-\tau^2]$ (if not clearly $\mathrm{LCE}(i,j) < \tau^2$) then we have that $i+\delta, j+\delta \in \mathcal{S}_\tau$ (Lemma 5). If $\mathrm{LCE}_R(i+\delta, j+\delta) \geq \delta$ it holds that $T[i...i+\delta] = T[j...j+\delta]$ so the algorithm correctly computes $\mathrm{LCE}(i,j)$ as $\delta + 1 + \mathrm{LCE}(i+\delta, j+\delta)$. Conversely, if $\mathrm{LCE}_R(i+\delta, j+\delta) < \delta$, $T[i...i+\delta] \neq T[j...j+\delta]$ it follows that $\mathrm{LCE}(i,j) < \delta < \tau^2$.

Query time is $\mathcal{O}(1)$, since computing δ, $\mathrm{LCE}_R(i+\delta, j+\delta)$ and $\mathrm{LCE}(i+\delta, j+\delta)$ all takes constant time. Storing the data structures for A and B takes space $\mathcal{O}(|A|+|B|) = \mathcal{O}(|\mathcal{S}_\tau|) = \mathcal{O}(\frac{n}{\tau})$. For the preprocessing stage, we can use recent algorithms by I et al. [8] for constructing the sparse suffix tree for A and B in $\mathcal{O}(\frac{n}{\tau})$ space. They provide a Monte Carlo algorithm using $\mathcal{O}(n \log \frac{n}{\tau})$ time (correct w.h.p.), and a Las Vegas algorithm using $\mathcal{O}(\frac{n}{\tau})$ time (w.h.p.).

4 Derandomizing the Monte Carlo Data Structure

Here we give a general technique for derandomizing Karp-Rabin fingerprints, and apply it to our Monte Carlo algorithm. The main result is that for any constant $\varepsilon > 0$, the data structure can be constructed completely deterministically in $\mathcal{O}(n^{2+\varepsilon})$ time using $\mathcal{O}(n/\tau)$ space. Thus, compared to the probabilistic preprocessing of the Las Vegas structure using $\mathcal{O}(n^{3/2})$ time with high probability, it is relatively cheap to derandomize the data structure completely.

Our derandomizing technique is stated in the following lemma.

Lemma 6. *Let $A, L \subset \{1, 2, \ldots, n\}$ be a set of positions and lengths respectively such that $\max(L) = n^{\Omega(1)}$. For every $\varepsilon \in (0, 1)$, there exist a fingerprinting function ϕ that can be evaluated in $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$ time and has the property that for all $a \in A, l \in L, i \in \{1, 2, \ldots, n\}$:*

$$\phi(T[a...a+(l-1)]) = \phi(T[i...i+(l-1)]) \iff T[a...a+(l-1)] = T[i...i+(l-1)]$$

We can find such a ϕ using $\mathcal{O}\left(\frac{S}{\varepsilon}\right)$ space and $\mathcal{O}\left(\frac{n^{1+\varepsilon}\log n}{\varepsilon^2}\frac{|A|}{S}\max(L)\,|L|\right)$ time, for any value of $S \in [1, |A|]$.

Proof. We let p be a prime contained in the interval $[\max(L)n^{\varepsilon}, 2\max(L)n^{\varepsilon}]$. The idea is to choose ϕ to be $\phi = (\phi_{p,x_1}, \ldots, \phi_{p,x_k})\,, k = \mathcal{O}(1/\varepsilon)$, where ϕ_{p,x_i} is the Karp-Rabin fingerprint with parameters p and x_i.

Let Σ be the alphabet containing the characters of T. If $p \leq |\Sigma|$ we note that since $p = n^{\Omega(1)}$ and $|\Sigma| = n^{\mathcal{O}(1)}$ we can split each character into $\mathcal{O}(1)$ characters from a smaller alphabet Σ' satisfying $p > |\Sigma'|$. So wlog. assume $p > |\Sigma|$ from now on.

We let $\mathbb{S}$ be the set defined by:

$$\mathbb{S} = \{(T[a \ldots a + (l-1)], T[i \ldots i + (l-1)]) \mid a \in A, l \in L, i \in \{1, 2, \ldots, n\}\}$$

Then we have to find ϕ such that $\phi(u) = \phi(v) \iff u = v$ for all $(u, v) \in \mathbb{S}$. We will choose $\phi_{p,x_1}, \phi_{p,x_2}, \ldots, \phi_{p,x_k}$ successively. For any $1 \leq l \leq k$ we let $\phi_{\leq l} = (\phi_{p,x_1}, \ldots, \phi_{p,x_l})$.

For any fingerprinting function f we denote by $B(f)$ the number of pairs $(u, v) \in \mathbb{S}$ such that $f(u) = f(v)$. We let $B(\mathrm{id})$ denote the number of pairs $(u, v) \in \mathbb{S}$ such that $u = v$. Hence $B(f) \geq B(\mathrm{id})$ for every fingerprinting function f, and f satisfies the requirement iff $B(f) = B(\mathrm{id})$.

Say that we have chosen $\phi_{\leq l}$ for some $l < k$. We now compare $B(\phi_{\leq l+1}) = B((\phi_{p,x_{l+1}}, \phi_{\leq l}))$ to $B(\phi_{\leq l})$ when x_{l+1} is chosen uniformly at random from $[p]$. For any pair $(u, v) \in \mathbb{S}$ such that $u \neq v$ and $\phi_{\leq l}(u) = \phi_{\leq l}(v)$ we see that, if x_{l+1} is chosen randomly independently then the probability that $\phi_{p,x_{l+1}}(u) = \phi_{p,x_{l+1}}(v)$ is at most $\frac{\max\{|u|,|v|\}}{p} \leq n^{-\varepsilon}$. Hence

$$\mathbb{E}_{x_{l+1}}(B(\phi_{\leq l+1}) - B(\mathrm{id})) \leq \frac{B(\phi_{\leq l}) - B(\mathrm{id})}{n^{\varepsilon}}\ ,$$

and thus there exists $x_{l+1} \in \{0, 1, \dots, p-1\}$ such that

$$B(\phi_{\le l+1}) - B(\mathrm{id})) \le \frac{B(\phi_{\le l}) - B(\mathrm{id})}{n^{\varepsilon}} .$$

Now consider the algorithm where we construct ϕ successively by choosing x_{l+1} such that $B(\phi_{\le l+1})$ is as small as possible. Then[4]

$$B(\phi) - B(\mathrm{id}) = B(\phi_{\le k}) - B(\mathrm{id}) \le \frac{B(\phi_{\le k-1}) - B(\mathrm{id})}{n^{\varepsilon}} \le \dots \le \frac{B(1) - B(\mathrm{id})}{n^{k\varepsilon}} .$$

Since $B(1) \le n^3$ and $B(\phi) - B(\mathrm{id})$ is an integer it suffices to choose $k = \left\lceil \frac{4}{\varepsilon} \right\rceil$ in order to conclude that $B(\phi) = B(\mathrm{id})$.

We just need to argue that for given $x_1, \dots, x_l$ we can find $B(\phi_{\le l})$ in space $\mathcal{O}\left(\frac{1}{\varepsilon}S\right)$ and time $\mathcal{O}\left(\frac{1}{\varepsilon}|L|\, n \log n \frac{|A|}{S}\right)$. First we show how to do it in the case $S = |A|$. To this end we will count the number of collisions for all $(u, v) \in \mathbb{S}$ such that $|u| = |v| = l \in L$ for some fixed $l \in L$. First we compute $\phi_{\le l}(T[a \dots a+(l-1])$ for all $a \in A$ and store them in a multiset. Using a sliding window the fingerprints can be computed and inserted in time $\mathcal{O}(ln \log n)$ Now for each position $i \in [1, n]$ we compute $\phi_{\le l}(T[i \dots i+(l-1])$ and count the number of times the fingerprint appears in the multiset. Using a sliding window this can be done in $\mathcal{O}(ln \log n)$ time as well. Doing this for all $l \in L$ gives the number of collisions.

Now assume that $S < |A|$. Then let $r = \left\lceil \frac{|A|}{S} \right\rceil = \mathcal{O}\left(\frac{|A|}{S}\right)$ and partition A into $A_1, \dots, A_r$ consisting of $\le S$ elements each. For each $j = 1, 2, \dots, r$ we define $\mathbb{S}_j$ as

$$\mathbb{S}_j = \{(T[a \dots a+(l-1)], T[i \dots i+(l-1)]) \mid a \in A_j, l \in L, i \in \{1, 2, \dots, n\}\}$$

Then $\mathbb{S}_1, \dots, \mathbb{S}_r$ is a partition of $\mathbb{S}$. For each $j = 1, 2, \dots, r$ we can find the collisions in among the elements in $\mathbb{S}_j$ in the manner described above. Doing this for each j and adding the results is sufficient to get the desired space and time complexity. □

Corollary 2. *For any $\tau \in [1, n]$, the LCE problem can be solved by a deterministic data structure with $\mathcal{O}(n/\tau)$ space usage and $\mathcal{O}(\tau)$ query time. The data structure can be constructed in $\mathcal{O}(n^{2+\varepsilon})$ time using $\mathcal{O}(n/\tau)$ space.*

Proof. We use the lemma with $A = \{k\tau \mid k \in [n/\tau]\}$, $L = \{2^l\tau \mid l \in [\log(n/\tau)]\}$, $S = |A| = n/\tau$ and a suitable small constant $\varepsilon > 0$.

References

1. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

[4] $B(1)$ is equal to $|\mathbb{S}|$ since 1 is the constant function

2. P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *J. of Discrete Algorithms*, 25:42–50, 2014.
3. D. Breslauer, R. Grossi, and F. Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013.
4. R. Cole and R. Hariharan. Approximate String Matching: A Simpler Faster Algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.
5. N. J. Fine and H. S. Wilf. Uniqueness Theorems for Periodic Functions. *Proc. AMS*, 16(1):109–114, 1965.
6. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69:525–546, 2004.
7. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
8. T. I, J. Kärkkäinen, and D. Kempa. Faster Sparse Suffix Sorting. In *Proc. 31st STACS*, volume 25, pages 386–396, Dagstuhl, Germany, 2014.
9. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Computing and Combinatorics, Second Annual International Conference, COCOON '96, Hong Kong, June 17-19, 1996, Proceedings*, pages 219–230, 1996.
10. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
11. O. Keller, T. Kopelowitz, and M. Lewenstein. Range non-overlapping indexing and successive list indexing. In *Proc. of Workshop on Algorithms and Data Structures (WADS)*, pages 625–636, 2007.
12. R. Kolpakov and G. Kucherov. Searching for gapped palindromes. In *Proc. 19th CPM, LNCS*, volume 5029, pages 18–30, 2008.
13. G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.
14. G. M. Landau and J. P. Schmidt. An Algorithm for Approximate Tandem Repeats. *J. Comput. Biol.*, 8(1):1–18, 2001.
15. G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *J. Algorithms*, 10:157–169, 1989.
16. H.-P. Lenhof and M. H. M. Smid. Using persistent data structures for adding range restrictions to searching problems. *Theoretical Informatics and Applications (ITA)*, 28(1):25–49, 1994.
17. M. Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302, 2013.
18. M. G. Main and R. J. Lorentz. An O (n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
19. G. Manacher. A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM*, 22(3):346–351, 1975.
20. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
21. E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
22. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th FOCS (SWAT)*, pages 1–11, 1973.